BSFuzzer: Context-Aware Semantic Fuzzing for BLE Logic Flaw Detection

Ting Yang[126], Yue Qin[3], Lan Zhang[4], Zhiyuan Fu[5], Junfan Chen[5], Jice Wang[5], Shangru Zhao[6], Qi Li[78*], Ruidong Li[2*], He Wang[1], and Yuqing Zhang[61*]
[1]School of Cyber Engineering, Xidian University, China
[2]School of Natural Science and Technology, Kanazawa University, Japan
[3]School of Information, Central University of Finance and Economics, China
[4]School of Informatics, Computing, and Cyber Systems, Northern Arizona University, USA
[5]College of Cyberspace Security, Hainan University, China
[6]School of Computer Science and Technology, University of Chinese Academy of Sciences, China
[7]Institute for Network Sciences and Cyberspace, Tsinghua University, China
[8]State Key Laboratory of Cryptology, China
yangt@nipc.org.cn, qinyue@cufe.edu.cn, Lan.Zhang@nau.edu, fuzy@hainanu.edu.cn, chenjunfan@hainanu.edu.cn, wangjice@hainanu.edu.cn, zhaosr@nipc.org.cn, qli01@tsinghua.edu.cn, lrd@se.kanazawa-u.ac.jp, hewang@xidian.edu.cn, zhangyq@ucas.ac.cn

***Abstract*—Bluetooth Low Energy (BLE) has become a foundational communication standard for modern connected devices. However, its complex design introduces subtle logic flaws, such as misinterpreted fields or invalid state transitions, that can enable authentication bypass, unauthorized control, or Denial-of-Service (DoS) attacks. These issues often evade conventional fuzzing and formal analysis. To address this gap, we propose BSFuzzer, a black-box, context-aware semantic fuzzing framework guided by the Bluetooth Core Specification. BSFuzzer uses a Large Language Model (LLM) agent to semantically parse the Bluetooth specification, extracting state machines and packet semantics from text, diagrams, and context. It then generates two types of mutations: field-level violations of protocol rules and state-level disruptions of key transitions. These are composed into structured test sequences and executed on target devices. The LLM agent is further used to verify responses against expected behaviors, enabling detection of subtle logic flaws beyond the reach of traditional fuzzers.**

We evaluated BSFuzzer on 19 real-world BLE devices, including 9 System-on-Chip (SoC) modules and 10 smartphones. It uncovered 36 security issues, including 34 previously unknown bugs, 9 of which have received CVE identifiers. Two critical flaws were recognized by a major vendor through bug bounty programs. The experimental results indicate that BSFuzzer attains high accuracy in both LLM-based specification analysis (up to 97%) and response validation (up to 85.8%), demonstrating its effectiveness in semantic extraction and enhancing fuzzing performance. Compared to four state-of-the-art BLE vulnerability detection tools, BSFuzzer achieved 9.34% higher code coverage and exposed a broader class of vulnerabilities, demonstrating its effectiveness in uncovering deep interpretation inconsistencies in BLE protocol implementations.

I. Introduction

The Bluetooth Low Energy (BLE) protocol, known for its low power consumption, high compatibility, and reliable connectivity, has become a cornerstone communication standard for a wide range of devices, including consumer wearables, medical monitoring devices, and Industrial Internet of Things (IIoT) systems [1]–[3]. The widespread adoption of BLE has significantly advanced device inter connectivity, but it has also exposed security vulnerabilities in increasingly diverse and critical application scenarios. Bluetooth-enabled devices suffer from diverse security flaws, ranging from privacy violations (e.g., BlueSnarfing [4], Badbluetooth [5]) and unauthorized device control (e.g., BLEEDINGBIT [6]) to Denial-of-Service (DoS) attacks (e.g., L2Fuzz [7]). As BLE adoption continues to expand, such security weaknesses pose escalating threats, with potential consequences ranging from personal data breaches to systemic failures in mission-critical applications.

Among the security weaknesses in BLE, logic flaws pose a uniquely critical and elusive threat due to their ability to bypass protocol-level safeguards without causing immediate, observable failures. By silently compromising authentication, encryption, or access control, they allow attackers to persistently undermine device integrity and exfiltrate sensitive data without raising alarms. Moreover, logic flaws are hidden within seemingly valid protocol interactions and are only triggered under specific sequences and contextual states, making their detection inherently non-trivial. *Therefore, to enhance the security of BLE-enabled systems, there is a pressing need for logic flaw detection methods that preserve protocol compliance and enable deep exploration of vulnerable states.*

Research Gap. Recent research has explored various techniques for identifying vulnerabilities in Bluetooth protocol implementations, including formal specification testing [8], state machine analysis [9], firmware emulation [10], and fuzzing [11]–[14]. Among them, fuzzing has emerged as the most widely adopted and practically effective approach for exposing memory corruption bugs via crash-triggering

* Corresponding authors: Yuqing Zhang, Qi Li, and Ruidong Li.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240094
www.ndss-symposium.org

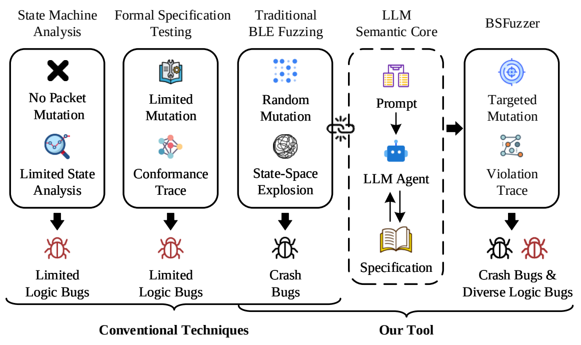


Fig. 1: LLM-Assisted BLE Fuzzing with Semantic and Contextual Knowledge.

malformed packets. However, these approaches remain limited in detecting logic flaws, which arise during valid protocol executions without causing crashes or observable failures. Detecting such flaws requires context-aware reasoning about state transitions, field constraints, and protocol-specified behavior, which are beyond the capabilities of traditional techniques. As a result, existing methods fall short in detecting subtle yet security-critical logic flaws in BLE implementations.

Our Solution: BSFuzzer. To bridge this gap, we present BSFuzzer, a context-aware semantic fuzzing framework designed to systematically detect logic vulnerabilities in BLE protocol implementations, as shown in Figure 1. Logic flaws arise when an implementation violates high-level protocol semantics—for example, by accepting an invalid field value such as MaxRxOctets = 20 (below the minimum allowed value of 27), or by permitting an out-of-order state transition, such as enabling encryption before completing authentication. These subtle violations often evade syntactic checks and detection, yet can silently compromise core security properties. A notable example is the BLUFFS attack [15], which exploited missing state-integrity checks during key derivation to compromise billions of Bluetooth devices. Motivated by these observations, we enhance fuzzing with the ability to detect logic flaws by extracting protocol semantics and contextual information from the Bluetooth Core Specification (hereafter referred to as the specification) and integrating them throughout the fuzzing pipeline. This protocol-aware approach enables the generation of semantically meaningful test cases that explore deeper protocol states while supporting precise, automated verification of device responses.

• *Challenges*. Achieving this capability involves overcoming two fundamental technical challenges. First, the specification encodes protocol semantics in a highly implicit and fragmented manner. Key rules are scattered across more than 3,000 pages of natural language descriptions, state diagrams, footnotes, and examples. Many critical constraints are not explicitly stated but are only implied through informal references or widely separated sections, which makes it difficult to extract a coherent and complete semantic model (**C1**). Second, even with a semantic understanding of the protocol, constructing test cases that reveal logic flaws requires carefully crafted multi-step message sequences. These sequences must satisfy all syntactic constraints while navigating through tightly constrained state transitions and field dependencies. Naive mutation or random permutation strategies fail to satisfy these conditions and often generate invalid or irrelevant inputs (**C2**). Together, these challenges underscore the need for a principled approach that combines semantic extraction, structured reasoning, and context-aware test synthesis to detect logic vulnerabilities effectively.

To address these challenges, BSFuzzer performs deep semantic parsing of the specification to extract the protocol's state machines and packet-level semantics. Instead of relying solely on sentence-level processing, the Large Language Model (LLM) operates at a broader scope by integrating natural language descriptions, state diagrams, and other contextual cues to infer packet semantics (**to address C1**). BSFuzzer uses the extracted semantic knowledge to generate two types of mutations: field mutation and state mutation. Field mutation leverages the semantic meaning of individual fields to deliberately violate protocol rules and trigger semantic inconsistencies. State mutation, on the other hand, exploits packet-to-packet dependencies to enforce contextual constraints and strategically inject mutations at critical points during protocol execution, thereby altering the state machine's behavior (**to address C2**). These mutated seeds are then composed into test sequences, which are transmitted to the System Under Test (SUT), and the corresponding responses are collected for analysis. In the bug verification stage, BSFuzzer invokes the LLM agent to extract behavioral handling strategies and continuously checks for deviations between expected and actual behaviors. Both inconsistencies and crash-level vulnerabilities are analyzed further, leading to detailed bug reports that uncover protocol flaws and security weaknesses.

We evaluated BSFuzzer on 9 BLE SoC devices and 10 smartphones, uncovering 36 bug instances, with 2 previously known. Of these, 9 were assigned CVEs, and two were acknowledged by the vendor with bug bounties. Compared to four state-of-the-art BLE vulnerability detection tools (Boofuzz [16], LLMIF [17], SweynTooth [12], and Proteus [8]), BSFuzzer discovered a broader range of bugs and achieved more comprehensive coverage[1]. Our method achieved 9.34% higher code coverage than Proteus, the strongest baseline. We evaluated the effectiveness of BSFuzzer across specification analysis, semantic seed generation, and response validation. In specification analysis, 112 field semantics and 595 packet dependencies were extracted with 92% and 97% accuracy, while 112 field-handling and 37 state-handling strategies achieved 94% and 85% accuracy, respectively. Semantic-guided mutations further improved bug discovery efficiency over random baselines. In response validation, the analyzer achieved an average accuracy of 85.8% for field-level and 74.0% for state-level validation. These results validate that BSFuzzer not only improves fuzzing depth and coverage, but also enables

[1]In this paper, unless otherwise specified, coverage refers to code coverage.

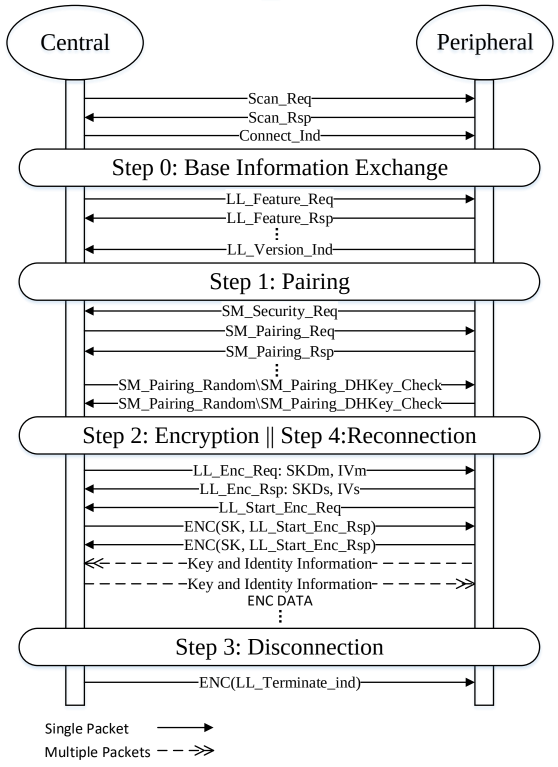


Fig. 2: The Messaging Flows in BLE Communication.

the discovery of previously undetected, high-impact flaws in real-world BLE stacks. Our contributions are summarized as follows:

- We designed BSFuzzer, a novel BLE fuzzing framework that operationalizes LLMs as semantic inference engines to interpret protocol intent and guide context-aware mutation, advancing the integration of semantic reasoning into systematic protocol fuzzing across multi-stage communication protocols.
- We proposed a specification-guided test sequence generator that leverages LLM-based reasoning to incorporate semantic constraints into field-level and state-level mutations, enabling the generation of state-aware, dependency-consistent test cases that effectively uncover logical vulnerabilities.
- We implemented and evaluated BSFuzzer on 19 real-world BLE devices, uncovering 34 previously unknown bugs (9 CVEs, two bug bounties from a major vendor). BSFuzzer is publicly available at: https://github.com/yangting111/BSFuzz.git.

II. Background and Motivation

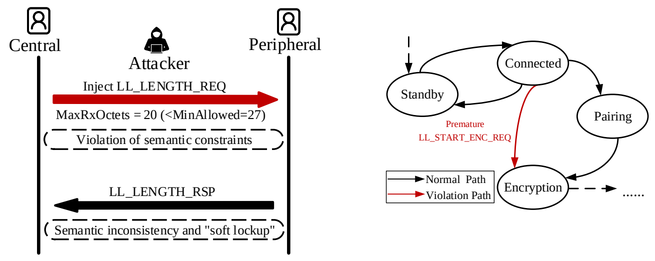
A. BLE Protocol

In BLE communication, devices typically take on one of two roles: central or peripheral. The central initiates and manages communication, while the peripheral responds to the central's commands and data requests. The basic BLE communication procedure is organized into multiple protocol phases, including device discovery, connection establishment, pairing, encryption, disconnection, and reconnection. These phases are illustrated in detail in Figure 2.

BLE Protocol Flow. The specification formalizes interaction sequences using Message Sequence Charts (MSCs), which visually represent the ordered sequence of message exchanges between the central and peripheral. Each MSC captures essential information such as the sender, receiver, message type, and temporal ordering of protocol events. Crucially, MSCs not only illustrate the normative state transitions that occur during standard operations but also define the expected behaviors under exceptional conditions, including authentication failures, timeout events, and aborted procedures. These formally described message flows serve as the foundation for constructing Finite State Machines (FSM) [18], which is widely used to model, analyze, and verify protocol behavior.

Protocol Semantic Information. In BLE, it refers to the higher-level meaning and rules that govern the expected protocol behavior beyond the raw byte-level layout of messages. It encompasses the intended purpose of each message and field, the conditions under which specific messages may be transmitted, the causal and conditional dependencies among messages across different protocol states, and the security and consistency rules that must be strictly enforced. For example, the `rand` and `ediv` fields carried in the `LL_ENC_REQ` message are not arbitrary values; rather, they are used to locate and validate the previously established Long Term Key (LTK) during the encryption setup process. These semantic rules form the critical foundation for BLE devices to operate correctly and securely, and understanding them is essential for effective protocol analysis and vulnerability detection.

Security Mechanisms. BLE is a layered communication protocol composed of multiple sub-protocols, including the Link Layer (LL), Logical Link Control and Adaptation Protocol (L2CAP), Attribute Protocol (ATT), Security Manager Protocol (SMP), and higher-level profiles, each responsible for distinct aspects of device interaction. The security mechanisms in BLE are implemented across multiple protocol layers, each serving a distinct role in enforcing different aspects of security. The LL layer ensures encryption and privacy through AES-CCM [19] and Resolvable Private Address (RPA)-based address resolution, preventing eavesdropping and tracking. The host layer, driven by the SMP, orchestrates the pairing process, negotiates authentication methods, and manages key distribution schemes, supporting both LE Legacy Pairing and LE Secure Connections. The SMP defines how devices exchange security capabilities and select an appropriate pairing method based on their I/O capabilities and security requirements. Once key exchange is completed, the LL enforces encryption at the transport level using algorithms such as AES-CCM to protect packet transmission integrity and confidentiality. Security mechanisms rely on the correct execution of protocol logic, but logic flaws break this assumption by enabling state manipulation or violations that cause security controls to fail or be bypassed.


Central
Attacker
Peripheral
Inject LL_LENGTH_REQ
MaxRxOctets = 20 (<MinAllowed=27)
Violation of semantic constraints
LL_LENGTH_RSP
Semantic inconsistency and "soft lockup"
Standby
Connected
Pairing
Encryption
Premature LL_START_ENC_REQ
Normal Path
Violation Path
......

a) Malformed LL Length Request. **b)** Encryption State Violation.

Fig. 3: Motivation Examples.

B. Motivation: Logic Flaws Beyond Conventional Detection

Despite advances in detection techniques such as formal specification testing [8], firmware emulation [10], and fuzzing [11]–[14], logic vulnerabilities—stemming from violations of protocol semantics and state machine constraints—remain largely unaddressed. These flaws are difficult to detect, as they rarely cause crashes and typically arise only during valid, context-sensitive protocol executions that current tools fail to analyze effectively. The risk is especially high in the mandatory LL and SMP, which govern tightly coupled procedures like connection setup, encryption, and authentication. Their complex, stateful interactions create fertile ground for subtle, security-critical flaws that conventional detection techniques are ill-equipped to expose. To illustrate these limitations and motivate our approach, we analyze two real-world cases uncovered during our research.

Case 1: State Inconsistency via Invalid MaxRxOctets Value. The specification explicitly defines the valid range of `MaxRxOctets` as 27 to 251 bytes, with the minimum value of 27 ensuring basic interoperability among devices. However, an attacker can exploit this by injecting an `LL_LENGTH_REQ` containing an invalid `MaxRxOctets` value below the required minimum (e.g., 20), to a vulnerable device, as shown in Figure 3a). If the peripheral device fails to enforce the lower-bound constraint strictly, it inadvertently accepts this invalid configuration, effectively downgrading its data reception capability to only 20 bytes. Although the connection appears normal initially, any subsequent legitimate packets exceeding 20 bytes are silently discarded. This condition creates interpretation inconsistencies and leads to a "soft deadlock", the device does not crash, yet effective communication becomes impossible.

Limitations of Existing Tools in Identifying This Vulnerability. The root cause of this bug stems from the BLE protocol stack's failure to strictly enforce the specified `MaxRxOctets` range and to execute the logic flow as defined by the specification. Insufficient validation of out-of-range values and inadequate error handling mechanisms allow devices to accept invalid configurations, leading to an overly restrictive receive buffer that silently discards legitimate packets exceeding the configured size. This disrupts normal communication without explicit error notifications. Contributing factors include inadequate input validation, deficient error handling routines, improper state management, and developers' incomplete or inconsistent interpretation of the specification. Existing approaches primarily monitor device feedback without reasoning about protocol logic. As a result, they cannot verify the correctness of responses to semantically invalid inputs and often miss logic-level flaws.

Case 2: State Machine Violation via Premature `LL_START_ENC_REQ`. According to the Bluetooth Core Specification, the encryption start request `LL_START_ENC_REQ` should only occur after successful pairing and encryption start procedures. However, in flawed implementations, if `LL_START_ENC_REQ` is sent immediately after establishing a connection, the peripheral device does not reject it due to incorrect state validation but instead prematurely transitions into encryption mode without negotiating a session key, as shown in Figure 3b). An attacker can exploit this flaw by prematurely injecting an `LL_START_ENC_REQ` packet right after connection establishment, forcing the device into an encrypted communication state. This premature transition leads to a mismatched encryption context, where the central and peripheral disagree on encryption status, ultimately resulting in failed communication and a connection timeout.

Limitations of Existing Tools in Identifying This Vulnerability. The root cause of this bug lies in the BLE peripheral device's improper implementation of state machine validation and insufficient input checks when processing the `LL_START_ENC_REQ` packet. This issue stems from developers' failure to strictly adhere to the Bluetooth Core Specification, including neglecting to enforce logic for rejecting unexpected packets and omitting robust state transition validations. Moreover, the flawed assumption that the central device would always conform to specification-compliant behavior resulted in insufficient defensive programming, allowing premature state changes without verifying the existence of a valid session key and ultimately leading to a mismatched encryption context. Existing tools, such as formal specification testing [8] and state machine analysis [9], cannot reliably generate the packet sequences required to reach vulnerable states or verify deviations from expected behavior, leaving them ineffective for detecting this class of vulnerabilities.

III. Design

A. Overview

The core idea of BSFuzzer is to integrate BLE protocol knowledge into the fuzzing process to enable the detection of semantic and logic flaws. By leveraging insights derived from the specification, BSFuzzer guides state tracking, enforces protocol constraints during test generation, and performs precise verification of device responses, thereby facilitating deeper, context-aware exploration of protocol behavior. Figure 4 shows the overall framework of BSFuzzer, which comprises three components: semantic parsing, test sequences generation, and bug verification. The input to BSFuzzer is the specification document. Specifically, (1) BSFuzzer begins by performing semantic parsing, which extracts two types of protocol knowledge from the specification: the state machine

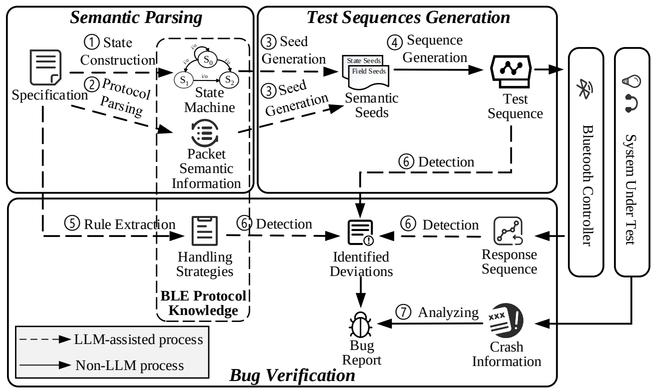
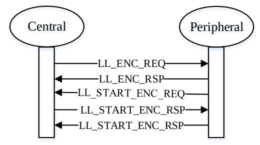


Fig. 4: Overview of BSFuzzer.



a) Message Sequence Chart.

Current State	Input/Output	Next State
S0	LL_ENC_REQ / LL_ENC_RSP, LL_START_ENC_REQ	S1
S1	LL_START_ENC_RSP / LL_START_ENC_RSP	S2

b) State Machine Extraction.

Fig. 5: State Machine Extraction of the *Start Encryption* Procedure on the Peripheral.

(derived from MSCs in step ①), and the packet semantic information (extracted by an LLM agent via protocol parsing in step ②). Together, these two outputs serve as the foundation for understanding protocol logic and behavioral constraints. (2) Based on this extracted semantic knowledge, BSFuzzer performs semantic seed generation. It produces field seeds (semantically meaningful input values) and state seeds (inputs designed to trigger specific transitions), as shown in step ③. These seeds are used in step ④ to synthesize context-aware test sequences that comply with the protocol's encryption, timing, and fragmentation rules. The generated sequences are then transmitted through a Bluetooth controller to the SUT, and the corresponding response sequences are collected. (3) In the bug verification stage, BSFuzzer uses the LLM agent to extract behavioral handling strategies (step ⑤) and continuously checks (step ⑥) for any identified deviations between the expected protocol behavior and actual device responses. Both deviations and crash artifacts are subjected to further analysis, enabling the generation of detailed bug reports that reveal potential security flaws.

B. Semantic Parsing

BSFuzzer leverages an LLM agent to extract protocol state machines and packet semantics from natural language, state diagrams, and scattered references, capturing key elements that are often implicit and dispersed throughout the specification. Below, we elaborate on the parsing process.

1) State Machine Construction: We parse the MSCs defined in the specification to extract the underlying protocol logic and construct the corresponding protocol state machines. For example, Figure 5 illustrates the state machine extraction process for the *Start Encryption* procedure. Figure 5a) shows the corresponding MSC, while Figure 5b) presents the extracted state machine from the perspective of the Peripheral. The message sequence is segmented into two macro-transitions. The first transition, from state S_0 to S_1, is triggered by the exchange of `LL_ENC_REQ`, `LL_ENC_RSP`, and `LL_START_ENC_REQ`, which represent the negotiation phase of the encryption setup. The second transition, from S_1 to S_2, involves the bidirectional exchange of `LL_START_ENC_RSP`, indicating mutual confirmation and completion of the encryption process. These state transitions are derived based on the temporal ordering and directionality of the messages in the MSCs. At each protocol state transition, we assign a label that records the expected input message and the corresponding output message during that transition by iterating over the MSCs defined in the LL and SMP procedures. We can construct a state machine that captures the normative behavior of the Peripheral during protocol execution.

2) Packet Semantic Information Parsing: BSFuzzer extracts two types of packet semantic information from the specification: single field semantics and packet-packet dependency, as illustrated in Figure 6.

Single Field Semantics. BSFuzzer employs a structured prompt (see Appendix A.1) that instructs the LLM agent to extract three key semantic attributes for each field: *Bit Length*, *Semantic Role*, and *Defined Values*. The *Bit Length* ensures structural correctness by enforcing the proper encoding size, preventing the generation of malformed packets that would be rejected by the target device. The *Semantic Role*, which describes the field's functional purpose within the protocol, is critical for modeling packet-packet dependency. The *Defined Values* capture valid ranges and special constants specified in the protocol, serving as the foundation for seed generation. For example, the `max_rx_bytes` field in the `LL_LENGTH_REQ` packet is a 16-bit value used to negotiate the maximum number of payload bytes a device can receive. The minimum valid value is explicitly defined as 0x001B, while the maximum is implicitly limited by the field's size to 0x00FB. BSFuzzer utilizes this information to ensure that the generated values strictly conform to the 16-bit format. Beyond structural compliance, BSFuzzer further exploits the *Defined Values* constraints to deliberately construct semantically invalid inputs, such as 0x001A.

Packet-Packet Dependency. In stateful protocols, a packet-packet dependency exists when a field ($f_1 \in P_1$) constrains the value or validity of a field ($f_2 \in P_2$), as dictated by protocol specifications or cryptographic mechanisms. Guided by a prompt (see Appendix A.2), the LLM agent extracts semantic links between fields in different packets using both textual definitions and normative constraints found in the specification. The prompt defines both the analysis input and the evaluation criteria, requiring the model to (1) strictly adhere to protocol-defined semantics, (2) detect only direct dependencies—where one field directly determines another,

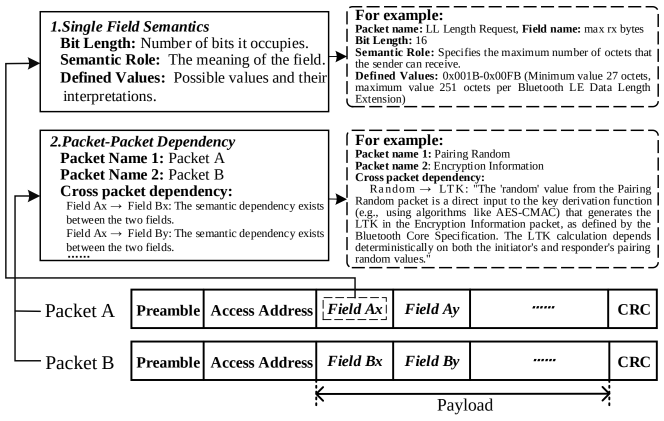

1.Single Field Semantics
Bit Length: Number of bits it occupies.
Semantic Role: The meaning of the field.
Defined Values: Possible values and their interpretations.
For example:
Packet name: LL Length Request, Field name: max rx bytes
Bit Length: 16
Semantic Role: Specifies the maximum number of octets that the sender can receive.
Defined Values: 0x001B-0x00FB (Minimum value 27 octets, maximum value 251 octets per Bluetooth LE Data Length Extension)
2.Packet-Packet Dependency
Packet Name 1: Packet A
Packet Name 2: Packet B
Cross packet dependency:
Field Ax → Field Bx: The semantic dependency exists between the two fields.
Field Ax → Field By: The semantic dependency exists between the two fields.
......
For example:
Packet name 1: Pairing Random
Packet name 2: Encryption Information
Cross packet dependency:
Random → LTK: "The 'random' value from the Pairing Random packet is a direct input to the key derivation function (e.g., using algorithms like AES-CMAC) that generates the LTK in the Encryption Information packet, as defined by the Bluetooth Core Specification. The LTK calculation depends deterministically on both the initiator's and responder's pairing random values."
Packet A
Preamble
Access Address
Field Ax
Field Ay
......
CRC
Packet B
Preamble
Access Address
Field Bx
Field By
......
CRC
Payload

Fig. 6: Example of Extracted Packet Semantic Information.

and (3) exclude indirect or speculative associations involving intermediate fields. For instance, as illustrated in Figure 6, the prompt enables the agent to identify a contextual dependency between the *Random* field in the `PAIRING_RANDOM` packet and the *LTK* field in the `ENCRYPTION_INFORMATION` packet, where the *Random* field directly influences the key derivation process according to the specification. More generally, by modeling interactions across protocol stages, BSFuzzer identifies dependencies where a field in a prior message constrains subsequent messages, capturing the semantic links governing valid protocol behavior.

C. Test Sequence Generation

Our approach uses specification-derived packet semantics and protocol states to generate test sequences that maintain valid context while deliberately violating selected constraints to trigger logic flaws. This ensures anomalies stem from intended violations rather than unrelated misconfigurations. BSFuzzer first generates semantic seeds, then constructs and sends complete packet sequences to the SUT. Below, we elaborate on the process of test sequence generation.

1) Seed Generation: We implement two mutation strategies: (1) field mutation, focusing on the SUT's handling of anomalous values within packet fields; and (2) state mutation, focusing on the SUT's ability to manage states throughout the entire protocol interaction flow.

Field Mutation. BSFuzzer employs a structured prompt (see Appendix A.3) that instructs the LLM agent to generate semantically guided mutations for a target field, enabling both semantic-logic testing and crash-bug exploration. The prompt defines the mutation scope and guides the process through five concrete mutation strategies: (1) Semantically Invalid Values: These values violate the protocol's defined value constraints as specified in the packet semantic information, which are used to evaluate how strictly an implementation adheres to protocol rules. (2) Bit-Length Boundary values: For each field, we generate boundary values based on its defined bit length. These values are useful for triggering edge-case behaviors. (3) Bit-Flip Mutations: Individual or multiple bits in the field are flipped to reveal improper masking or parsing errors in the implementation. (4) Length-Specific Structural mutations: For fields that define length semantics, we apply structural mutation strategies that intentionally create inconsistencies between the declared length and the actual data size. By crafting payloads that exceed or fall short of the specified length, these mutations induce parsing errors or unexpected behaviors. (5) Random Values: A controlled proportion of field values are randomly generated within the field's bit range. This introduces noise to cover a wider value space.

State Mutation. BSFuzzer employs a structured prompt (see Appendix A.4) that instructs the LLM agent to generate test paths around a target packet based on the specification-derived state machine (Section III-B1) and packet dependencies (Section III-B2). For each target packet, we analyze the specification to determine its state transition requirements (e.g., prerequisite conditions and valid state sequences) and, using packet dependency information, identify the critical protocol assumptions such as mandatory pre-state completions, restrictions on repeated operations, and constraints on permissible state transitions. Based on these dependencies and the baseline state machine, we construct packet transmission paths that deliberately violate at least one of these assumptions. Each mutation path includes the target packet and falls into one or more of the following categories: (1) Prestate Not Completed: sending a packet before its prerequisite states or dependent packets have been satisfied; (2) Repeated Operations: re-sending a state-altering packet already processed, thereby violating one-time-use constraints; (3) Unexpected State Change: forcing the system into an invalid or undefined state by manipulating the order of state transitions; (4) Combined Violations: combining two or more of the above violation types to achieve deeper coverage of potential logic flaws.

2) Sequence Generation: This part is built upon two core components: a *Mapper*, which instantiates abstract protocol packets, and a *Cryptographic Calculation Module*, which ensures compliance with protocol-specific security requirements. Together, these components enable the construction of context-aware test inputs, forming the foundation for executing both field mutation and state mutation strategies.

Mapper. Responsible for converting abstract representations of protocol messages, derived from protocol state machines or state mutation strategies, into fully structured binary packet formats. During this process, the *Mapper* populates each field based on configuration templates and dependency constraints, ensuring syntactic correctness and contextual relevance. As illustrated in Figure 7, the abstract message ω is translated into

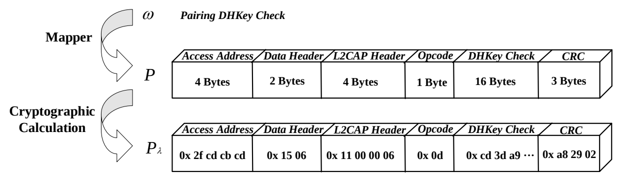


Fig. 7: Transformation of Abstract `PAIRING_DHKEY_CHECK` into Concrete BLE Packet.

a structured packet format P, which is a hierarchical structure composed of multiple layers, with each layer containing fields that represent specific attributes of the BLE protocol.

Cryptographic Calculation Module. This component is responsible for computing values in security-critical fields to ensure that the generated packets conform to protocol security requirements. At the LL, it handles real-time encryption and decryption in coordination with key management. In addition, segmentation is applied to satisfy physical-layer transmission constraints, ensuring that packet construction reflects realistic protocol behavior. At the SMP layer, it executes cryptographic algorithms to derive key material and validate the authenticity of exchanged messages. As illustrated in Figure 7, this module processes the structured packet P and computes session-specific cryptographic fields to produce the final transmission-ready packet P_λ. The parameters required to compute the `PAIRING_DHKEY_CHECK` value, such as random numbers, pairing credentials, and role identifiers, are dynamically generated during the pairing process to reflect the session-specific security context.

The *Sequence Generation* process is adapted according to the mutation strategy in use: For field mutation, we follow a valid protocol execution path as defined by the state machine. Abstract packets are instantiated into concrete formats using the *Mapper*. Along this execution path, we identify the target packet for mutation. When the protocol flow reaches this packet, the designated field is modified to the mutation value. Throughout this process, the message sequence, session state, encryption status, and other contextual parameters remain unchanged, ensuring that any observed behavioral differences are attributed solely to the mutated field. For state mutation, BSFuzzer operates at the message-sequence level. The entire packet sequence is constructed based on mutated control logic derived from a state seed. Each packet in the mutated sequence is then instantiated using the *Mapper* and processed through the *Cryptographic Calculation Module* to ensure semantic and cryptographic correctness.

D. Bug Verification

BSFuzzer extracts field-level and state-level handling rules from the specification and employs an LLM agent to validate device responses against expected behaviors, revealing improper field handling and state inconsistencies. It combines automated checks with targeted analysis to efficiently detect logic flaws in critical transitions and unexpected acceptances. We elaborate on the verification process as below.

1) Handling Rule Extraction: BSFuzzer supports two protocol validation types: field-level, which targets violations in individual message fields, and state-level, which detects inconsistencies caused by invalid protocol states, as shown in Figure 8.

Field-Level Rule Extraction. BSFuzzer employs a prompt (see Appendix A.5) that instructs the LLM agent to analyze the field semantics and derive the corresponding handling rules. This prompt guides the model through two main components:

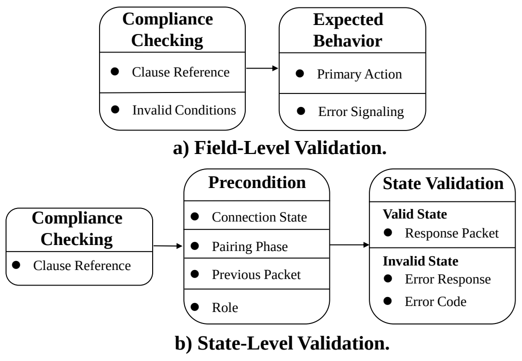


Fig. 8: Handling Rule Extraction.

(1) *Violation Scenarios*, which distinguish between protocol-level violations, where field values directly violate format or range constraints, and capability mismatches, where the values are technically valid but exceed the implementation's supported limits; (2) *Behavior Prediction*, where the model identifies the mandatory behavior required by the specification, any recommended behavior for optional handling, and the expected error signaling actions—such as silent drop, sending an error code, or issuing a reject packet. This structured extraction allows BSFuzzer to map field-level violations to their expected protocol responses, enabling precise compliance checking against the normative specification.

State-Level Rule Extraction. BSFuzzer uses a prompt template (see Appendix A.6) designed to extract the required preconditions under which each protocol message may be processed. This prompt guides the model through two main components: (1) *Precondition Extraction*, where the model identifies the contextual requirements under which a given protocol message is allowed to be processed. These preconditions are based on factors such as the stack layer, device role, packet type, packet direction, and security mode. The model determines whether the current device state satisfies these normative conditions defined in the specification. (2) *Behavior Prediction*, where the model infers the expected response behavior based on the state validation result. If the preconditions are satisfied, the model identifies the correct next-step behavior as defined by the specification. If the preconditions are violated, the model extracts the mandated error-handling behavior, including the type of rejection packet and the corresponding error code required by the protocol. This structured extraction allows BSFuzzer to validate state-dependent protocol logic and detect semantic inconsistencies that arise from processing packets in invalid contexts, enabling comprehensive state-level compliance checking.

2) Inconsistency Detection: This module leverages an LLM agent to compare device responses with expected protocol behaviors, thereby identifying inconsistencies in protocol implementations. It adopts two targeted prompt designs:

Field-Level Validation. We focus on the handling of anomalous values in individual fields of BLE control packets (see the field validation prompt in Appendix A.7). (1) The LLM agent first determines whether the mutated field values comply with the Bluetooth Core Specification. (2) If the field value is invalid, it further checks whether the device's response adheres to the protocol-defined error-handling logic.

State-Level Validation. We target interaction sequences consisting of multiple BLE packets (see the state validation prompt in Appendix A.8). (1) The LLM agent verifies, packet by packet, whether the device is in a valid protocol state to legally receive the current packet. (2) If the precondition state is valid, the LLM agent checks whether the device's response follows the expected state transition; if the precondition state is invalid, it checks whether the device correctly returns the corresponding error response as specified.

3) Bug Analysis: We manually analyze each logged inconsistency to trace its root cause and determine whether it originates from implementation defects or protocol misinterpretations. To minimize the amount of manual analysis required, we designed distinct verification rules for field mutations and state mutations.

Field-Level Analysis. We use the expected behaviors defined in the Bluetooth Core Specification to assess whether the device's responses are correct. Because mutated packets are deliberately crafted to violate protocol rules, rejecting them is typically considered normal. Verification, therefore, focuses only on cases where invalid inputs are unexpectedly accepted. If the device correctly rejects a mutated input, no further action is required. However, if it unexpectedly accepts the input, the corresponding sequence is subjected to further analysis.

State-Level Analysis. These usually involve packet sequences designed to violate the expected state machine behavior. We focus only on critical state transitions. First, we identify packets capable of triggering significant protocol state changes and exclude those that do not lead to meaningful transitions (e.g., `LL_VERSION_IND` in the Link Layer). Verification then checks whether these critical state transitions occur as expected. This selective focus on a small number of high-risk events significantly reduces the need for extensive manual analysis.

IV. Implementation

We have implemented BSFuzzer in Python 3 [20] using a modular approach to facilitate future extensions. We ran our tool on Ubuntu 22.04.4 LTS, 16GB RAM, Intel Core i7-13700F CPU, 500GB Disk, and nRF52840 dongle [21]. We utilized the driver developed in the SweynTooth [12] and flashed it to the nRF52840 dongle, enabling the dongle to send and receive raw LL packets to and from the BLE device.

LLM Usage. We develop an LLM agent system powered by Grok-3 [22], capable of semantic understanding and reasoning over protocol specifications. The specification is first modularized by sub-protocol, and each module is embedded into vector representations using OpenAI's text-embedding-3-large model [23]. These embedded modules serve as callable external knowledge sources, enabling the agent to perform semantic retrieval, interpretation, and alignment of normative content.

Semantic Parsing. We modeled the BLE protocol as Mealy machines and generated DOT files for both the Legacy Pairing and Secure Connections Pairing methods. To ensure that the extracted data is accurate and factual, we set the LLM temperature to 0 when parsing structured information from the BLE specifications. This parsing process is a one-time effort. From the specifications, we extracted semantic information for the LL and SMP layers, covering 84 fields across 22 LL packets and 34 fields across 14 SMP packets. Based on these field semantics, we identified 48 pairs of packets with dependency relationships, including 22 intra-LL dependencies, 17 intra-SMP dependencies, and 9 cross-layer dependencies between LL and SMP.

Test Sequence Generation. BSFuzzer mutates state-machine traces extracted from the specification and subsequently validated on real devices. Mutation operators are derived from specification-defined semantic dependencies and interaction patterns, each encoding a specific intent and applying controlled changes to field constraints or state transitions. Using an LLM with temperature set to 1, we generated diverse and semantically valid seeds for both Legacy and Secure Connections pairing. These seeds were executed by a 2K-line Python sequence engine that constructs protocol-consistent test flows, resolving dependencies and handling cryptographic parameters across key BLE interactions, including connection establishment, capability negotiation, encryption initiation, and reconnection, to achieve semantic coverage of real-world protocol behaviors.

Bug Verification. To validate protocol compliance, we again used the LLM with the temperature set to 0. To reduce manual analysis effort, we implemented over 300 lines of Python code to define filtering rules, thereby further narrowing down the results requiring human verification.

V. Evaluation

To evaluate the effectiveness of our tool, we conducted experiments on real-world BLE protocol implementations. Our evaluation aims to answer the following questions:

Q1. Does our tool achieve better detection of logic vulnerabilities than existing BLE fuzzers? (Section V-B)

Q2. How does each component of our design contribute to discovering logic vulnerability? (Section V-C)

Q3. Can our tool uncover previously unknown bugs in real-world BLE protocol implementations? (Section V-D)

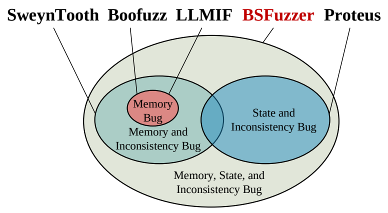
A. Experimental Setup

1) Real-world BLE devices: To evaluate the generalizability and robustness of our approach, we selected 9 different BLE SoCs from 9 distinct vendors (such as Cypress [24], Microchip [25]) and 10 smartphones from 7 major smartphone manufacturers (such as Google [26], Huawei [27]), ensuring broad coverage of widely used BLE platforms and popular commercial brands. As summarized in Appendix B.1, these devices span various BLE versions and are representative of mainstream deployments. The selected BLE SoCs are widely integrated into products across multiple industries, including

Device	Boofuzz			LLMIF			SweynTooth			Proteus			BSFuzzer		
	M	**S**	**I**	**M**	**S**	**I**	**M**	**S**	**I**	**M**	**S**	**I**	**M**	**S**	**I**
CY8CKIT-042-BLE	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1
WBZ451	1	0	0	1	0	0	1	0	0	0	0	0	1	1	1
ESP32-WROOM-32E	1	0	0	1	0	0	1	0	1	0	1	1	1	2	2
RTL8762EKF-EVB	1	0	0	1	0	0	1	0	0	0	1	0	1	3	0
SUM	3	0	0	3	0	0	4	0	0	0	2	1	4	7	4

Note: The table shows the number of discovered bugs in three categories: **M** (memory corruption), **S** (state bugs), and **I** (inconsistency bugs).

a) Bugs Found on Different Devices.


SweynTooth
Boofuzz
LLMIF
BSFuzzer
Proteus
Memory Bug
Memory and Inconsistency Bug
State and Inconsistency Bug
Memory, State, and Inconsistency Bug

b) Bug Categories across Tools.

Fig. 9: Comparison of Bugs Discovered by Different Fuzzing Tools.

smart home systems, wearable devices, medical equipment, and industrial automation. Each SoC-based device was programmed using the sample code provided by its corresponding Software Development Kit (SDK), and by modifying the configuration files, we customized key parameters such as pairing methods and authentication modes. For smartphone-based evaluation, we leveraged nRF Connect to configure the mobile devices as servers and enable advertising, allowing us to flexibly set different device modes and conduct experiments.

2) Baseline Methods: We select four fuzzing tools as our baseline: Boofuzz [16], LLMIF [17], SweynTooth [12], and Proteus [8]. These tools are either published in top security conferences or widely recognized in the community. Boofuzz and LLMIF are general-purpose black-box fuzzing tools for IoT protocols, while SweynTooth and Proteus are dedicated fuzzers for BLE protocol implementations. Boofuzz is widely adopted in the security industry for protocol fuzz testing; in our evaluation, we use Boofuzz to define the structure of BLE packets and adopt its default mutation strategies as a baseline for comparison. LLMIF integrates LLM into the fuzzing process and, although originally designed for Zigbee protocol testing, its prompt-based architecture is adaptable to BLE. We leverage LLMIF's prompt templates for BLE seed generation and response verification as part of our comparative analysis. SweynTooth is a BLE-specific fuzzer that applies particle swarm optimization to discover bugs. It focuses on detecting memory-related crashes and has successfully identified a set of such flaws in BLE protocol stacks from several major vendors. In contrast, Proteus adopts a model-guided approach using protocol state machines and predefined properties to uncover state-related logical bugs in BLE implementations.

B. Comparison with Existing Works

To evaluate the effectiveness of our approach, we compare it against existing baseline tools across three key dimensions: qualitative comparison, bug detection capability and coverage growth over time.

1) Qualitative Comparison: As summarized in Table I, we compare our tool with baseline works across five qualitative dimensions. Among these tools, Boofuzz and SweynTooth do not leverage any form of protocol specification knowledge to guide test case generation. Instead, they adopt traditional fuzzing strategies based on generic input mutation, which limits their ability to generate semantically valid or state-sensitive inputs. In contrast, LLMIF, Proteus, and our tool incorporate specification-aware logic. Notably, only our tool supports fine-grained semantic analysis at both the field level and the state level. SweynTooth, Proteus, and our tool support stateful fuzzing, which enables the preservation and manipulation of internal BLE protocol states during test execution. This capability allows these tools to explore deeper protocol logic and trigger state-dependent bugs. Regarding bug detection diversity, baseline tools such as Boofuzz, LLMIF, and SweynTooth primarily identify memory-related bugs. Proteus expands this scope by including state logic errors and specification inconsistencies. Proteus demonstrates improved bug exposure through state and inconsistency analysis, but lacks support for semantic consistency verification, which is critical for identifying complex bugs stemming from interactions between protocol layers.

TABLE I: Comparison with Baseline Tool.

Tool	Specification Awareness	State Awareness	Semantic Awareness Mutation	Semantic Consistency Verification	Bug Detection Diversity
Boofuzz	×	×	×	×	M
LLMIF	✓	×	✓	×	M
SweynTooth	×	✓	×	×	M/I
Proteus	✓	✓	✓	×	S/I
Our tool	✓	✓	✓	✓	M/S/I

Note: **M** (memory corruption), **S** (state bugs), and **I** (inconsistency bugs).

2) Bug Detection Capability: We selected four BLE SoCs as benchmarks and compared our approach against existing baseline works. We present a detailed comparison of the number and types of bugs discovered during a 24-hour fuzzing session, as shown in Figure 9a) and Figure 9b). Boofuzz and LLMIF are not specifically designed for BLE protocols and therefore lack support for BLE-specific state transitions, making them incapable of exploring deep protocol logic and uncovering state-dependent bugs. SweynTooth, while effective in detecting certain implementation flaws, is unable to identify state-dependent issues due to the absence of appropriate test oracles. Proteus's detection is limited to predefined properties in the BLE Core Specification, identifying only explicitly specified violations while missing deep logical vulnerabilities requiring understanding of protocol states, field semantics, and behavioral logic. In contrast, our approach leverages protocol semantics-aware test generation and enables semantic consistency verification of test cases and responses to reveal hidden inconsistencies, potentially uncovering bugs overlooked by existing methods.

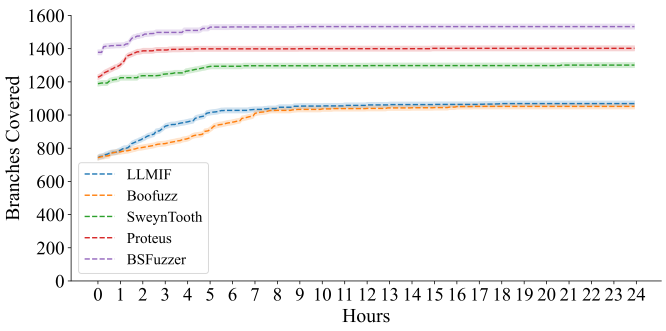


Fig. 10: Branch Coverage Over Time for Different Fuzzing Tools.

3) Coverage Growth Over Time: The objective of fuzzing is to maximize the activation of distinct logical execution paths within the target implementation. Higher code coverage indicates that the method more effectively explores protocol logic, state transitions, and boundary conditions, thereby facilitating the discovery of deep logical flaws. Since our approach targets the LL and SMP sub-protocols as integral components of the full protocol stack, we adopt coverage growth over time as the evaluation metric and compare our results against existing baseline tools. To evaluate the code coverage of our approach, we compare the results against existing baseline tools. However, since most BLE SoCs do not open-source their protocol stack implementations, existing evaluations are typically limited to black-box testing without access to internal coverage metrics. We follow Proteus's methodology by running the open-source `btstack` and collecting coverage data via `gcov` every five minutes. We repeated the entire fuzzing campaign three independent times, and the observed variation among runs was very small (standard deviation < 0.018). As shown in Figure 10, our method achieves a 9.34% increase in code coverage compared to Proteus, which was previously reported to achieve the highest coverage among existing tools.

The improvement in code coverage is related to the protocol depth explored by our fuzzing tool. LE Legacy Pairing and Secure Connections Pairing represent two distinct authentication and key establishment procedures, each involving different message sequences, cryptographic computations, and state transitions. By supporting both pairing modes and the associated encryption-handling logic, our approach is capable of triggering a diverse set of code paths. In addition, our field mutation and state mutation mechanisms enable the precise triggering of edge-case behaviors and state-dependent logic within the protocol implementation. As a result, our tool exercises deeper protocol logic, thereby achieving broader code coverage.

C. Effectiveness

1) Effectiveness of LLM-based Specification Analysis: We first analyzed early LLM extraction results to identify common hallucination and omission patterns. Based on these observations, we introduced structured output constraints and explicit specification references to guide the generation process. The prompts were iteratively refined through measured accuracy and detailed error analysis on manually annotated samples, with each iteration re-evaluated until the accuracy converged to a stable level. During extraction, every artifact is validated across multiple runs, and any inconsistency or reference violation triggers targeted manual review. In practice, all 22 LL and 10 SMP Message Sequence Charts were correctly transformed into protocol state machines; 112 field semantics and 595 packet dependencies achieved accuracies of 92% and 97%, respectively; and 112 field-handling and 37 state-handling strategies reached accuracies of 94% and 85%. The entire analysis, encompassing state machine derivation, semantic extraction, and handling strategy identification, is performed once as a preprocessing step. The resulting structured artifacts are cached and reused across all fuzzing runs, while manual verification requires only about 4.5 hours in total.

2) Effectiveness of Semantic Seed Generation: To evaluate the effectiveness of semantic seed generation in triggering protocol bugs, we conducted a comparative study on two BLE devices, comparing semantic seeds with randomly generated seeds under two mutation strategies: *field mutation* and *state mutation*. Each strategy includes a semantic-aware variant and a random baseline:

- For *field mutation*,
  - *Fs* uses field values guided by protocol semantics,
  - *Fr* relies on randomly generated field values.
- For *state mutation*,
  - *Ss* generates state transitions guided by analyzing protocol dependencies,
  - *Sr* performs random state transitions without semantic consideration.

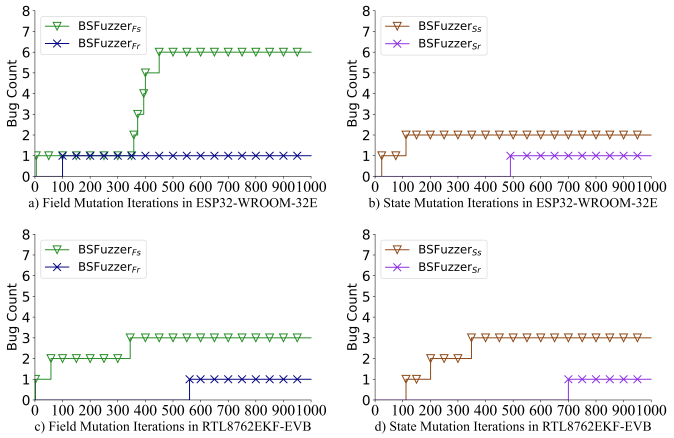


Fig. 11: Effectiveness of Semantic vs. Random Seed Generation under Field and State Mutation Strategies.

As shown in Figure 11, BSFuzzer$_{Fs}$ is more effective at producing edge cases and malformed inputs, leading to faster and more consistent discovery of bugs and inconsistencies.

TABLE II: Details of Discovered Bugs.

ID	Bug Description	Pair.	Issue	CVSS	Security Impact Summary	Spec. Ref.
M1	Buffer overflow via oversized `len` in `LL_PAUSE_ENC_REQ`	Leg.	DoS	6.5	Causes crash due to unchecked field size	–
M2	Buffer overflow via RFU=7 in `LL_TERMINATE_IND`	–	DoS	4.3	Improper handling of reserved value leads to memory fault	–
M3	Invalid `ChMap`=0x0 in `CONNECT_REQ`	–	DoS	6.5	Triggers crash during setup, preventing new connections	V6B§4.5.8
M4	Invalid `AccessAddr`=0x0 in `CONNECT_REQ`	–	DoS	6.5	Triggers crash during setup, preventing new connections	V6B§2.1.2
M5	Invalid `timeout`=0xFFFF in `CONNECT_REQ`	–	DoS	6.5	Triggers crash during setup, preventing new connections	V6B§4.5.2
M6	Invalid `win_offset`=0xFFFF in `CONNECT_REQ`	–	DoS	6.5	Triggers crash during setup, preventing new connections	V6B§2.3.3
S1	Accepts `LL_PAUSE_ENC_REQ` before encryption is enabled	–	DoS	7.5	Allows early transition to encryption pause state	V6B§4.7
S2	SK reuse allows reconnection without re-pairing	SC	SecByp	8.8	Enables attacker to rejoin session using old SK	V3H§2.3.5
S3	Accepts zero LTK after `PAIRING_FAILED`	Leg.	SecByp	8.1	Neglects key validation; encryption proceeds with all-zero key	V3H§2.4.4
S4	Misaligned state by early `LL_ENC_REQ` before pairing	–	DoS	7.5	Improper sequence handling leads to desync	V3H§2.4
S5	Accepts `LL_START_ENC_RSP` before pairing	–	DoS	8.1	Fails to verify preconditions before encryption start	V6B§4.7
S6	Multiple `LL_LENGTH_REQ` causes crash	–	DoS	6.5	Flooding `LL_LENGTH_REQ` causes stack overflow	V6B§4.5.10
S7	Accepts `LL_START_ENC_REQ` before pairing	–	DoS	7.5	Bypasses authentication phase during pairing	V6B§4.7
S8	Accepts `PAIRING_RANDOM` before public key exchange	SC	DoS	7.5	Breaks pairing sequence	V3H§2.3.5.7
I1	Accepts `LL_LENGTH_REQ` with `max_tx_bytes` < 27	–	DoS	6.5	Violates minimal payload constraint	V6B§4.5.10
I2	Falls back to Legacy despite SC request	SC	Downg.	7.6	Allows downgrade to weaker authentication mode	V3H§2.2
I3	Incorrect response to malformed `LL_PAUSE_ENC_REQ` with invalid RFU	–	–	–	–	V6B§4.7
I4	Incorrect response to malformed `LL_LENGTH_REQ` with invalid params	–	–	–	–	V3H§2.3.5.10
I5	Disconnect triggered by premature `LL_PAUSE_ENC_REQ`	–	DoS	4.5	State inconsistency during encryption setup	V6B§4.7

Note: Leg.=Legacy Pairing; SC=Secure Connections; DoS=Denial of Service; SecByp=Security Bypass; Downg.=Security Downgrade. "–" means no observable security impact. "V6B§4.7" denotes Volume 6, Part B, Section 4.7 of the specification.

TABLE III: Validation Results of the Accuracy of the LLM-based Bug Analyzer.

Pairing Type	Mutation Type	Total Inputs	Correct Cases	False Positive	False Negative	Accuracy
Legacy Pairing	Field	1000	854	112	34	85.4%
Legacy Pairing	State	500	369	125	6	73.8%
Secure Connections	Field	1000	862	110	28	86.2%
Secure Connections	State	500	371	120	9	74.2%

Similarly, BSFuzzer$_{Ss}$ navigates the protocol state space more efficiently by avoiding invalid paths and prioritizing meaningful state transitions. In contrast, BSFuzzer$_{Fr}$ often produces invalid packets, resulting in longer bug-triggering time and wasted communication resources. BSFuzzer$_{Sr}$ exhibits shallow exploration and frequently encounters unreachable or illegal states, making it less effective in uncovering deep logic flaws.

3) Effectiveness of LLM-based Response Validation: To evaluate the effectiveness of the LLM-based bug analyzer in validating device responses, we conducted an experiment involving both Secure Connections and Legacy Pairing. For each pairing mode, we manually analyzed the results produced by the LLM agent under two mutation strategies: field mutation and state mutation. As shown in Table III, the LLM agent achieved over 85% accuracy in validating responses from field mutations. In contrast, its accuracy was lower for state mutations, primarily due to the inherent complexity of reasoning over protocol state transitions. Manual validation of each response is time-consuming and often requires examining thousands of specification pages. In contrast, the LLM agent can automatically reference specifications, significantly improving the efficiency of identifying suspicious responses. As demonstrated in Table III, while LLM-based analysis is not the sole basis for bug detection and may introduce false negatives and false positives, it remains highly effective at filtering high-probability anomalies. False negatives have minimal impact since vulnerable test sequences typically appear multiple times during fuzzing, ensuring abnormal behavior can still be detected. False positives only increase manual verification overhead without affecting overall detection accuracy.

D. Discovered Real-world Bugs

We categorize the identified bugs into three types: memory corruption, state bugs, and inconsistency bugs, as shown in Appendix B.2. Across the 19 tested devices, we discovered a total of 36 bug instances. Given the wireless nature of BLE communication, the overall fuzzing time is influenced by connection latency and signal stability. Therefore, we measure the duration of a single fuzzing iteration from the moment the connection is established to the point when the last test packet is sent. The time taken for the device to manifest abnormal behavior or for the bug to be observed is not included in this measurement, similar to SweynTooth [12]. The *Disclosure Status* column in Appendix B.2 summarizes the current disclosure progress of each bug. All identified bugs have been disclosed to vendors. Nine bugs (two memory corruption, five state bugs and two inconsistency bugs) have been officially assigned CVE identifiers. To eliminate redundancy, we conducted necessary-condition tests for each Proof of Concept (PoC) to identify the essential fields or packets that trigger the bug and merged cases sharing the same root cause. After deduplication, these instances correspond to 19 unique root causes, including 6 memory corruption, 8 state, and 5 inconsistency bugs, as shown in Table II. Detailed descriptions are provided below.

Memory Corruption: M1 and M2. Both vulnerabilities stem from improper handling of mutated control packet fields, resulting in buffer overflows. M1 is triggered by an oversized length field in the `LL_PAUSE_ENC_REQ` packet. When the payload exceeds expected bounds, the device writes beyond allocated memory, causing a crash. This indicates insufficient bounds checking during control packet parsing. M2 arises from setting a reserved RFU field to 0x07 in the `LL_TERMINATE_IND` packet. According to the Bluetooth specification, RFU bits must be ignored by the receiver; however, the device incorrectly parses them, leading to memory corruption due to unsafe field handling.

Memory Corruption: M3, M4, M5 and M6. These vulnerabilities are caused by the device's inability to properly handle

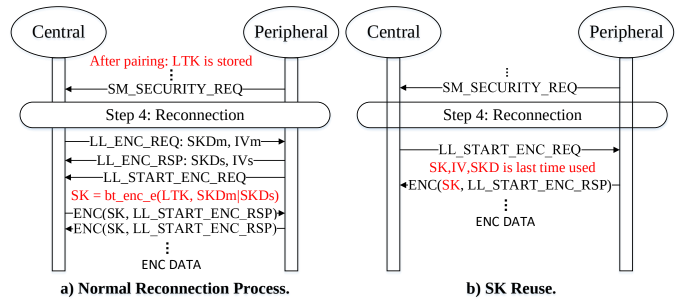


Fig. 12: An Illustration of Session Key Reuse.

invalid values in critical fields during connection establishment, leading to crashes due to memory corruption. M3 is triggered by a CONNECT_IND packet with a *Channel Map* set to 0x0000000000, which disables all 37 data channels. According to the specification, at least one data channel must be enabled; violating this constraint results in undefined behavior during channel selection. M4 involves the use of an all-zero *Access Address*, which is prohibited by the specification and leads to errors during connection filtering. M5 is caused by setting the *connSupervisionTimeout* field to 0xFFFF, an explicitly invalid value outside the allowed range of 0x000A to 0x0C80 (100 ms to 32.0 s). Finally, M6 results from assigning *winOffset* = 0xFFFF, which exceeds the maximum legal value of 0x1FFF (12.799 ms). In all cases, the lack of value validation results in a crash upon receiving malformed connection parameters.

State Bug: S1, S5 and S7. These vulnerabilities stem from improper handling of encryption control procedures, causing devices to enter encryption states at invalid times. S1 occurs when the device accepts an LL_PAUSE_ENC_REQ before encryption is established. S5 and S7 involve premature entry into the encrypted state: S5 is triggered by receiving an LL_START_ENC_RSP, and S7 by an LL_START_ENC_REQ, both before pairing. According to the specification, these procedures are valid only after encryption is properly initiated. Accepting these packets prematurely leads to state desynchronization, which can block legitimate communication, cause undefined behavior, or enable DoS attacks. These bugs arise from missing encryption-state validation in the LL state machine.

State Bug: S2. The device allows reconnection using a previously stored session key without requiring re-pairing. According to the specification, during the reconnection process, the device is expected to retrieve the LTK from the previous bonding context and combine it with a newly exchanged Session Key Diversifier (SKD) to derive a temporary Session Key (SK) for AES-CCM encryption, ensuring freshness and forward secrecy, as illustrated in Figure 12a). However, the device deviated from this expected behavior and directly resumed encrypted communication using the old SK without re-executing key derivation, as illustrated in Figure 12b). The root cause lies in the device's failure to enforce key diversification and bonding state validation during re-connection.

State Bug: S3. The device accepts encryption requests using an all-zero LTK after receiving a PAIRING_FAILED message, bypassing the required pairing procedure. This violates the SMP state machine and enables unauthenticated encrypted communication. As a result, the connection becomes vulnerable to unauthorized access. The root cause is a state machine flaw in handling the PAIRING_FAILED message, which allows the device to skip the pairing phase entirely.

State Bug: S4. The device receives an LL_ENC_REQ before pairing and correctly rejects it, but the connection remains active. An attacker can subsequently send an LL_PAUSE_ENC_REQ, placing the device into an inconsistent encryption state that causes all subsequent packets to be ignored. This results in communication failure and can be exploited to launch a DoS attack. The root cause is a state machine flaw that permits encryption-related procedures to proceed despite a prior encryption rejection, without validating the current security state.

State Bug: S6. The device sends multiple LL_LENGTH_REQ packets during the pairing process, leading to a crash. It fails to properly serialize control procedure handling across the LL and SMP, resulting in memory corruption. The root cause lies in a cross-layer state management flaw, where the device fails to enforce mutual exclusion or ordering between ongoing pairing operations and lower-layer control exchanges.

Inconsistency Bug: I1. The devices (D4, D7, D11, D14 and D15) accept LL_LENGTH_REQ packets where *max_tx_bytes* or *max_rx_bytes* are set below the minimum allowed value of 27 bytes. According to the specification, such values are invalid and must be rejected. An attacker can exploit this behavior to inject malformed packets during communication, leading to connection failure or DoS. The devices (D1, D10, and D17) accept fragmented data packets with payload lengths below the minimum of 27 bytes. This behavior may lead to unstable communication. The root cause is insufficient input validation during the data length update procedure.

Inconsistency Bug: I2. The device falls back to Legacy Pairing even though both peers advertise and require Secure Connections. According to the specification, if both devices support *SC* (as indicated by the *SC* bit in the *authReq* field of the PAIRING_REQUEST and PAIRING_RESPONSE packets), the pairing procedure must enforce Secure Connections. This downgrade violates the pairing requirements negotiation protocol and weakens the security guarantees of the connection. The root cause lies in the incorrect enforcement of the authReq flags during the pairing method selection process.

Inconsistency Bug: I3 and I4. These two bugs are caused by improper handling of malformed control packet fields, resulting in inconsistent protocol behavior. For I3, according to the specification, *RFU* fields must be ignored by the receiver, and the packet should be processed as if the *RFU* bits were set to zero. However, the device incorrectly attempts to parse these bits, leading to unexpected responses or silent discards. For I4, the issue arises when the device accepts an LL_LENGTH_REQ packet containing invalid parameters. The specification requires devices to accept only valid values; however, devices D2 and D3 incorrectly respond to such malformed requests with an `LL_LENGTH_RSP` instead of rejecting them, violating the expected protocol behavior.

Inconsistency Bug: I5. This bug is triggered when the device receives an `LL_PAUSE_ENC_REQ` before encryption is established, causing unexpected connection termination. According to the specification, this control procedure is valid only during an active encrypted session and should be ignored or rejected if received prematurely. However, the device instead treats it as a fatal error and disconnects, indicating a violation of protocol robustness requirements. This behavior can be exploited for DoS attacks. The root cause is insufficient encryption-state validation before handling control procedures, leading the device to process the packet in an invalid context.

VI. Discussion and Limitations

Device Hardware Constraints. Due to hardware limitations of the SUT, our evaluation of the pairing process was limited to the *Just Works* authentication method. Alternative methods such as *Out of Band* and *Passkey Entry* were not tested. While these methods only affect the authentication phase and do not alter the core encryption or authentication logic of the protocol, potential vulnerabilities specific to them may have gone undetected. Future work could incorporate a broader range of authentication mechanisms to provide a more comprehensive assessment of security features in BLE implementations.

Handling Non-Structured Packets. BSFuzzer conducts packet dependency analysis by leveraging the semantic meanings of individual protocol fields. However, several packets in the SMP layer lack structured fields. For instance, the `LL_START_ENC_REQ` packet is an LL control Protocol Data Unit that contains only an *Opcode* field without any additional parameters. This absence of fields limits the applicability of our standard prompt templates for dependency extraction. Consequently, such cases require special handling and manual analysis to infer their relationships with other packets.

Manual Effort. The reliability of the LLM-based verifier tends to decline when analyzing complex protocol state transitions, which often involve dependencies and contextual information spanning multiple interaction steps, with certain conditions even requiring consideration of historical states or implicit protocol semantics to correctly interpret device behavior. In such scenarios, the LLM agent may produce misjudgments or overlook latent issues. Nevertheless, the verifier's outputs remain highly valuable during bug analysis, enabling researchers to rapidly pinpoint suspicious interaction sequences or responses, significantly narrow the analysis scope, and reduce the need to manually examine large volumes of protocol logs line by line.

VII. Related Work

Bluetooth Stack Security Testing. The security issues in the BLE stack have drawn significant attention. Some studies have already been conducted to identify potential vulnerabilities in the BLE stack. SweynTooth [12] designs a BLE fuzzing framework that achieves full control over communication at the LL to identify vulnerabilities in BLE protocol implementations. BLEDiff [9] utilizes an active automata learning approach to extract the FSM of a BLE implementation. A checking module is then used to detect deviations from expected protocol behavior. Proteus [8] proposes a state machine mutation-based testing framework to uncover logical vulnerabilities in wireless protocol implementations. Pferscher *et al.* [13] implemented a stateful black-box fuzzing from the connection procedure to the start of the pairing procedure of BLE devices using automata learning. BrakTooth [14] enables fuzz testing by modifying the central's requests. However, they may not identify flaws in the program logic of protocol implementations. BrokenMesh [28] conducted fuzz testing on the network build and network control stages of the BLE Mesh protocol to assess the security of their implementation. Some of the work conducted fuzz testing only on specific functions of the Bluetooth stacks from certain manufacturers. Sami Babigeon *et al.* [29] adopted a fuzzing approach to test the OTA function of the GATT service on Silicon BLE devices. Matias Karhumaa [30] tested the controller of Zephyr's Bluetooth LE stack by fuzzing. Zinuo Han [31] found multiple vulnerabilities by fuzzing and code auditing the Bluetooth protocol implementation in AOSP. Some studies focus on fuzz testing specific subprotocols within the Bluetooth protocol. ToothPicker [11] implemented an in-process Bluetooth daemon fuzzer based on Frida to evaluate the implementation of the Bluetooth protocol in the iOS system. L2Fuzz [7] implements a stateful fuzzer to detect vulnerabilities in the Bluetooth BR/EDR L2CAP layer. Frankenstein [10] fuzz the Bluetooth stack of Broadcom and Cypress firmware in an emulated environment.

LLM-based Network Protocol Security Testing. Recent advances have explored the use of LLMs to enhance various aspects of protocol security testing. These studies leverage the language understanding and reasoning capabilities of LLMs to automate traditionally manual tasks such as specification parsing [32], seed generation [17], [33], and state inference [34]. ChatHTTPFuzz [32] utilizes LLMs to annotate data fields of the HTTP protocol [35], construct structured seed templates, and identify optimal mutation points. ChatAFL [33] combines LLMs with public RFCs to enrich initial seeds for the RTSP protocol [36], enabling structure-aware mutation. LLMIF [17] integrates LLMs with the ZigBee specification [37] to assist in seed generation, mutation, and test case evaluation. mGPTFuzz [34] employs LLMs to extract state machines from the Matter protocol [38] to guide fuzzing, while LLMgSSA [39] leverages LLM reasoning to infer valuable protocol states during fuzzing. MBFuzzer [40] further adopts an LLM-driven approach to analyze bug reports and validate protocol non-compliance. In contrast to these works, BSFuzzer moves beyond token-level or format-based mutation and operationalizes LLMs as semantic inference engines capable of comprehending protocol state machines and inter-packet dependencies. This enables context-aware mutations, targeted state transitions, and dependency-preserving field perturbations that expose logic flaws beyond existing LLM-based fuzzers. Moreover, because mutation operators are grounded in semantic and dependency representations extracted from formal specifications, the methodology is generalizable to other multi-stage communication protocols beyond Bluetooth.

VIII. Conclusion

In this paper, we presented BSFuzzer, a context-aware semantic fuzzing framework for BLE protocol stacks that leverages protocol knowledge extracted from specifications. Specifically, BSFuzzer first performs semantic parsing, which combines both state machine information and semantic information extracted by an LLM agent. Based on this knowledge, it generates mutation seeds, including field seeds and state seeds, to create context-aware test cases. By executing these cases and validating responses against expected semantic behavior, BSFuzzer effectively detects logic and state inconsistencies invisible to traditional fuzzers. We evaluated BSFuzzer on 19 real-world BLE devices, uncovering 34 previously undocumented bugs, including memory corruption, semantic violations, and state transition flaws. Among these, 9 bugs received CVE identifiers, and two were acknowledged by the vendor with bug bounties. BSFuzzer outperformed four baseline tools, achieving 9.34% higher code coverage, demonstrating its effectiveness in exposing high-impact, context-sensitive bugs in BLE implementations.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (U2336203, 62132011), the National Key Research and Development Program of China (2023QY1202), the Beijing Natural Science Foundation (L242015, 4242031), the Innovation Fund of Xidian University (GNYZ2024QC009), the Beijing-Tianjin-Hebei Natural Science Foundation Cooperation Project (No. 25JJJJC0003), and the China Scholarship Council (CSC) International Cooperation Training Program for Innovative Cybersecurity Talents (CXXM20210118).

References

[1] Bluetooth SIG, "Bluetooth® market update 2025," https://www.bluetooth.com/2025-market-update/, 2025, accessed: 2025-06-23.

[2] Statista, "Bluetooth – topic overview," Statista Topic Page, 2025, https://www.statista.com/topics/7730/bluetooth/#topicOverview.

[3] IndustryARC, "Bluetooth low energy market size report 2024–2030," IndustryARC market report, 2024, https://www.industryarc.com/Report/187/bluetooth-smart-market-forecast.html.

[4] A. Agarwal, "Bluebugging: How hackers utilize bluetooth-enabled devices to steal data," 2023, times of India Blog. [Online]. Available: https://timesofindia.indiatimes.com/blogs/voices/bluebugging-how-hackers-utilize-bluetooth-enabled-devices-to-steal-data/

[5] F. Xu, W. Diao, Z. Li, J. Chen, and K. Zhang, "Badbluetooth: Breaking android security mechanisms via malicious bluetooth peripherals." in *NDSS*, 2019.

[6] B. Seri and G. Vishnepolsky, "Bleedingbit: The hidden attack surface within ble chips," *Black Hat USA*, Dec. 2018, presented at Black Hat USA 2019; accessed via Black Hat EU 2018 materials. [Online]. Available: https://i.blackhat.com/eu-18/Thu-Dec-6/eu-18-Seri-BleedingBit-wp.pdf

[7] H. Park, C. K. Nkuba, S. Woo, and H. Lee, "L2fuzz: Discovering bluetooth l2cap vulnerabilities using stateful fuzz testing," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022, pp. 343–354.

[8] S. M. M. Rashid, T. Wu, K. Tu, A. A. Ishtiaq, R. H. Tanvir, Y. Dong, O. Chowdhury, and S. R. Hussain, "State machine mutation-based testing framework for wireless communication protocols," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2102–2116.

[9] I. Karim, A. Al Ishtiaq, S. R. Hussain, and E. Bertino, "Blediff: Scalable and property-agnostic noncompliance checking for ble implementations," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 3209–3227.

[10] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 19–36.

[11] D. Heinze, J. Classen, and M. Hollick, "ToothPicker: Apple picking in the iOS bluetooth stack," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/heinze

[12] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurniawan, "{SweynTooth}: unleashing mayhem over bluetooth low energy," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 911–925.

[13] A. Pferscher and B. K. Aichernig, "Stateful black-box fuzzing of bluetooth devices using automata learning," in *NASA Formal Methods Symposium*. Springer, 2022, pp. 373–392.

[14] M. E. Garbelini, V. Bedi, S. Chattopadhyay, S. Sun, and E. Kurniawan, "{BrakTooth}: Causing havoc on bluetooth link manager via directed fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1025–1042.

[15] D. Antonioli, "Bluffs: Bluetooth forward and future secrecy attacks and defenses," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 636–650.

[16] B. Community, "Boofuzz: Network protocol fuzzing for humans," https://github.com/jtpereyda/boofuzz, accessed: 2024-10-22.

[17] J. Wang, L. Yu, and X. Luo, "Llmif: Augmented large language model for fuzzing iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 881–896.

[18] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001.

[19] Morris Dworkin, "NIST Special Publication 800-38C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality," National Institute of Standards and Technology (NIST), Tech. Rep. SP 800-38C, 2007. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38c.pdf

[20] Python Software Foundation, "Python 3.11.0 documentation," https://docs.python.org/3.11/, 2022, accessed: 2025-05-04.

[21] Nordic Semiconductor, "nRF52840 Dongle," https://www.nordicsemi.com/Products/Development-hardware/nRF52840-Dongle, 2025, accessed: 2025-05-04.

[22] xAI, "xai api documentation," https://docs.x.ai/docs/overview, 2025, accessed: 2025-06-20.

[23] OpenAI, "text-embedding-3-large," https://platform.openai.com/docs/guides/embeddings, 2024, https://platform.openai.com/docs/guides/embeddings/embedding-models.

[24] Cypress Semiconductor Corporation, "Cy8ckit-042-ble-a bluetooth low energy pioneer kit guide," Cypress Semiconductor Corporation, User Guide 002-11468 Rev. E, 2018, https://www.infineon.com/dgdl/Infineon-CY8CKIT-042-BLE-A_Bluetooth_Low_Energy_Pioneer_Kit_Guide-UserManual-v01_00-EN.pdf.

[25] Microchip Technology Inc., "Wbz451 curiosity board user's guide," Microchip Technology Inc., User Guide DS50003367, 2022, https://ww1.microchip.com/downloads/aemDocuments/documents/WSG/ProductDocuments/UserGuides/WBZ451-Curiosity-Board-User-Guide-DS50003367.pdf.

[26] Google LLC, "Pixel 6 pro – learn about your pixel," Google product page, 2025, https://pixel.withgoogle.com/Pixel_6_Pro?hl=en&country=US. [Online]. Available: https://pixel.withgoogle.com/Pixel_6_Pro?hl=en&country=US

[27] Huawei Device Co., Ltd., "Huawei phones," Huawei Smartphone official product page, 2025, https://consumer.huawei.com/ke/phones/.

[28] D. K. Han Yan, Lewei Qu. (2024) Brokenmesh: New attack surfaces of bluetooth mesh. [Online]. Avail-

able: https://www.blackhat.com/us22/briefings/schedule/#brokenmesh-new-attack-surfaces-of-bluetooth-mesh-26853

[29] B. F. Sami Babigeon. (2023) Breaking secure boot on the silicon labs gecko platform. [Online]. Available: https://blog.quarkslab.com/breaking-secure-boot-on-the-silicon-labs-gecko-platform.html

[30] M. Karhumaa. (2021) Cyrc vulnerability advisory: Denial-of-service vulnerabilities in zephyr bluetooth le stack. [Online]. Available: https://www.synopsys.com/blogs/software-security/cyrc-advisory-zephyr-vulnerability.html

[31] Z. Han. (2022) Deep into android bluetooth bug hunting - new attack surfaces and weak code patterns. [Online]. Available: https://i.blackhat.com/EU-22/Thursday-Briefings/EU-22-Zinuo-Deep-into-Android-Bluetooth-Bug-Hunting.pdf

[32] Z. Yang, H. Peng, Y. Jiang, X. Li, H. Du, S. Wang, and J. Liu, "Chathttpfuzz: large language model-assisted iot http fuzzing," *International Journal of Machine Learning and Cybernetics*, pp. 1–22, 2025.

[33] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[34] X. Ma, L. Luo, and Q. Zeng, "From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter {IoT} devices," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4783–4800.

[35] M. Belshe, R. Peon, and M. Thomson, "Hypertext transfer protocol version 2 (http/2)," RFC 7540, IETF, May 2015, https://datatracker.ietf.org/doc/html/rfc7540.

[36] H. Schulzrinne, A. Rao, and R. Lanphier, "Real time streaming protocol (rtsp)," RFC 2326, Internet Engineering Task Force (IETF), April 1998, https://datatracker.ietf.org/doc/html/rfc2326.

[37] ZigBee Alliance, "Zigbee specification," ZigBee Document 053474r17, 2008, https://zigbeealliance.org/solution/zigbee/.

[38] Connectivity Standards Alliance, "Matter specification, version 1.4," Connectivity Standards Alliance Standard, 2024, https://csa-iot.org/wp-content/uploads/2024/11/24-27349-006_Matter-1.4-Core-Specification.pdf.

[39] B. Yu, Q. Song, and C. Cai, "Large language model guided state selection approach for fuzzing network protocol," in *2024 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, 2024, pp. 1–6.

[40] X. Song, J. Wu, Y. Zeng, H. Pan, C. Zuo, Q. Zhao, and S. Guo, "Mbfuzzer: A multi-party protocol fuzzer for MQTT brokers," in *Proceedings of the USENIX Security Symposium*. USENIX Association, 2025, implementation and evaluation report a prototype that triggered 73 bugs across six mainstream MQTT brokers.

Appendix A
Prompt Implementation

Prompt Template for Field Semantics Extraction

As a Bluetooth protocol expert, please perform the following protocol analysis task: Analysis Target: The **[Field]** in the **[Packet]**
Execution Steps:
1) Locate Packet Structure Accurately identify the definition of the **[Packet]** in the protocol document. Parse the binary structure of the **[Field]**.
2) Field Semantic Analysis Extract the intended purpose and functional description of the field.
3) Value Range Derivation
a. Basic Value Range:
Determine the bit-length of the field.
Calculate the theoretical minimum (0) and maximum $2^n - 1$ values.
b. Semantic Value Range:
Extract descriptions of reserved bits.

[Example Output]

Fig. A.1: Prompt Template for Field Semantics Extraction.

Prompt Template for Packet-Packet Dependency

As a Bluetooth protocol expert, you need to analyze the direct dependencies between the two data packet parameters of **[Packet_1]** and **[Packet_2]**.
Please follow these steps for professional analysis:
Data packet definition:
ALL **[Packet_1]** field: **[Packet_1_fields]**
ALL **[Packet_2]** field: **[Packet_2_fields]**
Analysis requirements:
1. Strictly analyze based on the provided **[Packet_1]** fields and **[Packet_2]** fields without omitting or fabricating any fields.
2. Direct dependency is defined as: The parameter value in one data packet directly affects or determines the setting of the parameter value in another data packet.
3. Consider the constraints between the parameters defined by the protocol specification.
4. Exclude indirect effects or associations generated by intermediate parameters.
Complete it in the <think> tag:
1. List the protocol definition of each parameter
2. Analyze the direct causal relationship between the parameters
3. Record the excluded indirect associations
Output requirements:
1. Use strict JSON format
2. Contain Boolean cross_packet_dependency fields
3. Describe each parameter pair relationship with key-value pairs in the explanation field
4. Keeping the same field structure as the example

[Example Output]

Fig. A.2: Prompt Template for Packet-Packet Dependency.

Prompt Template for Field Mutation

You are a Bluetooth protocol testing expert specializing in fuzz testing for Bluetooth peripheral, given the following packet and field information, generate mutations targeting this field.

Background Information: **[Field Semantics]**
Field Mutation: boundary conditions, semantic invalid cases, bit-flip tests, random values
[Example Output]

Fig. A.3: Prompt Template for Field Mutation.

Prompt Template for State Mutation

As a Bluetooth protocol testing expert, your task is to generate protocol mutation test cases as central role based on state dependencies.
Please follow the process below:
Target Data Packet:
[Packet]
[Dependent packets]
[Dependency info]
[State transition path]

Generation Rules:
1.Parse Dependent packets to get the state transition requirements related to **[Packet]**.
2.Comprehensive coverage of **[Packet]**-related protocol violations to ensure comprehensive testing.
3.Based on the state transition requirements, generate the packet transmission path, at least one of the following state dependencies must be violated:
-Must include: **[Packet]**
-Prestate Not Completed
-Repeated Operations Not Following Protocol Specifications
-Unexpected State Change
-Combinations containing the above variants
-Each test case corresponds to a specific failure scenario
-Technical basis description (cite the protocol chapter or specification clause)

Generation Steps:

1. List violations of specific parts and descriptions related to **[Packet]** in `<Analysis>` tags.
2. Generate the packet transmission path.
3. Only focus on the **[Packet]** related state transition mutation.
3. Confirm in the `<Validation>` tag whether each mutation directly violates protocol clauses.
4. Describe the specific mutation steps in `<Mutation Steps>`, using only: - Send [data packet name] - Skip [data packet name].
5. Generate xx test cases, no repeat.

[Example Output]

Fig. A.4: Prompt Template for State Mutation.

Prompt Template for Field Validation Rules

As a Bluetooth Core Specification expert, analyze the expected device behavior upon receiving a packet with invalid or unsupported field values and a correct CRC. Follow the structure below:
[Single Field Semantics]
Analysis Requirements
1. Violation Scenarios
Protocol-Level Violation: Directly violates the defined value range or format.
Capability Mismatch: Technically valid, but exceeds receiver's implementation limits.

2. Behavior Prediction
Mandatory Behavior: Required behavior per the spec
Recommended Behavior: Optional implementation
behavior Error Signaling: Silent drop / Send error code / Return reject packet, prioritization of error codes.

[Example Output]

Fig. A.5: Prompt Template for Field Validation Rules.

Prompt Template for State Validation Rules

As a Bluetooth Core Specification certified expert, analyze the required preconditions for the specified packet within the protocol stack and predict the device's response behavior based on whether the state is valid or invalid.
Protocol Scope:
Layers: [LL,SMP]
Role: peripheral,
Security mode: **[Device Security Mode]**,
Packet Type: **[packet_name]**,
Direction: Received by peripheral device

[Example Output]

Fig. A.6: Prompt Template for State Validation Rules.

Prompt Template for Field Validation

You are an expert in the Bluetooth Core Specification. A single field in a BLE control packet has been mutated: **[Packet_mutation]** **[Actual_response]**, where "empty" in **[Actual_response]** indicates a silently dropped packet. Perform the following steps:
1. Field Validity Verification
Determine whether the mutated field value is valid or invalid according to the **[Specification_Section]**.
2. Device Response Compliance
If the field value is invalid, evaluate whether the device's actual response aligns with the expected behavior **[Expected_Device_Behavior]**.

[Example Output]

Fig. A.7: Prompt Template for Field Validation.

Prompt Template for State Validation

You are a Bluetooth Core Specification expert. Given a sequence of BLE packets: **[Send_Sequence]** and **[Device_Response_Sequence]**, perform the following checks for **[Send_Packet]** in the path:

1. Pre-State Verification

For **[Send_Packet]**, determine whether the device was in a valid protocol state to legally receive and process this packet, based on the **[Precondition]**.

If the send packet is expected in the current state, it should be treated as a state-valid packet.

If the send packet is unexpected in the current state, it should be treated as a state-invalid packet.

2. Device Response Compliance

For response packet:

If the pre-state was valid, verify whether the device replied with the expected **[Valid_Response]**.

If the pre-state was invalid, verify whether the device replied with the expected **[Invalid_Response]**.

[Example Output]

Fig. A.8: Prompt Template for State Validation.

APPENDIX B
EXPERIMENTAL DEVICES AND FINDINGS

TABLE B.1: List of BLE Devices.

Category	ID	Device	Vendor/Manufacturer	BLE Ver.	BLE SDK/System Ver.	Sample Code
BLE SoC	D1	CY8CKIT-042-BLE	Cypress	5.1	V3.66	BLE_Battery_Level
	D2	WBZ451	Microchip	5.2	V1.3.0	BLE_Throughput
	D3	ESP32-WROOM-32E	Espressif	4.2	V5.3.0	GATT_Security_Server
	D4	B91	Telink	5.0	V4.0.1.0	Tlkapp_General
	D5	RTL8762EKF-EVB	Realtek	5.0	V1.4.0	BLE_Peripheral
	D6	CH592	WCH	5.3	V1.8	Peripheral
	D7	LP-CC2652RB	Texas Instruments	5.2	V7.41.00.17	Throughput_Peripheral
	D8	nRF52840-DK	Nordic Semiconductor	5.0	V2.9.1	Peripheral_Hids_Mouse
	D9	NUCLEO-WB55RG	STMicroelectronics	5.2	V1.21.0	BLE_HeartRateThreadX
Smartphone	D10	P40	Huawei	5.1	HarmonyOS 4.2.0	/
	D11	Nova5	Huawei	5.0	HarmonyOS 4.0.0	/
	D12	Mate50	Huawei	5.2	HarmonyOS 4.2.0	/
	D13	Honor90	Honor	5.2	Android 13	/
	D14	OPPO A72	OPPO	5.0	ColorOS V12.1	/
	D15	Note 10 Pro	Redmi	5.0	MIUI 13.0.13	/
	D16	Xiaomi 14	Xiaomi	5.4	HyperOS 2.0.60	/
	D17	Galaxy C55	Samsung	5.2	Android14	/
	D18	Google Pixel 2	Google	5.0	Android 10	/
	D19	Google Pixel 6 Pro	Google	5.2	Android 15	/

TABLE B.2: Summary of Bugs Found on the Tested Devices.

Device	Bugs			Fuzzing Time	Disclosure Status
	M	**S**	**I**		
D1	M1	S3	I1	1h 38m	Two CVE Assigned
D2	M6	S5	I4	3h 26m	Vendor Reporting
D3	M4	S1,S2	I3,I4	2h 08m	S2:CVE Assigned; CVE Request
D4	-	-	I1	2h 57m	Vendor Reporting
D5	M2	S6,S7,S8	I1	1h 20m	Four CVE Assigned
D6	-	S1	I2	1h 46m	Vendor Reporting
D7		S1	I5	3h 02m	Two CVE Assigned
D8	-	-	-	2h 23m	-
D9	M5	-	-	2h 21m	Vendor Reporting
D10	-	S2	I1	2h 25m	Bug Bounty
D11	-	S2	I1	2h 03m	Bug Bounty
D12	-	-	I1	2h 11m	I1:Bug Bounty,Vendor Reporting
D13	-	-	-	2h 05m	-
D14	M3	S1	I1	3h 16m	M3:Previously Reported,Vendor Reporting
D15	M3	S1	I1	1h 47m	M3:Previously Reported,Vendor Reporting
D16	M3	-	-	1h 28m	Vendor Confirmed
D17	-	-	I1	1h 22m	Vendor Confirmed
D18			I5	1h 38m	Vendor Reporting
D19	-	S8	-	1h 27m	Vendor Reporting

Note: **M** (memory corruption), **S** (state bugs), and **I** (inconsistency bugs).